DOI: https://doi.org/10.25140/2411-5363-2025-2(40)-312-324
УДК 004.41

Олексій Сергійович Кондус[1], Олексій Миколайович Ткаченко[2]

[1]магістр
Київський національний університет імені Тараса Шевченка (Київ, Україна)
E-mail: oleksii.kondus@knu.ua. **ORCID:** https://orcid.org/0009-0003-2514-6196. **ResearcherID:** NLN-6147-2025

[2]кандидат технічних наук, доцент кафедри теорії та технології програмування
Київський національний університет імені Тараса Шевченка (Київ, Україна)
E-mail: otkachenko@knu.ua. **ORCID:** https://orcid.org/0000-0002-9514-516X. **ResearcherID:** U-3613-2017

МОДУЛЬ ІНТЕЛЕКТУАЛІЗОВАНОЇ ПІДТРИМКИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка програмного забезпечення включає ряд рутинних процесів, які потребують різної кількості ресурсів, залежно від специфіки задач. У багатьох випадках це впливає на продуктивність команди і дотримання графіку робіт. У статті запропоновано програмне рішення – Smart AI Code Assistant – розширення для Visual Studio Code, що забезпечує прискорення розробки шляхом автоматизації деяких рутинних процесів за допомогою AI-моделей, зокрема, GPT, Claude, Gemini, Grok та DeepSeek. На відміну від існуючих рішень, розроблений модуль дозволяє вибрати модель залежно від задачі, проаналізувати код на відповідність принципам S.O.L.I.D., інтегрувати функції генерування документації, тестів, рефакторингу та пояснення логіки коду в єдиному інтерфейсі. Експериментально підтверджено ефективність моделей у задачі рефакторингу коду.

Ключові слова: *автоматизація розробки; штучний інтелект; S.O.L.I.D.; рефакторинг; документація коду; AI-моделі; Visual Studio Code.*

Рис.: 12. Табл.: 1. Бібл.: 12.

Актуальність теми дослідження. В умовах високої конкуренції на ринку ІТ швидкість виходу продукту і якість програмного забезпечення (ПЗ) є ключовими факторами успіху. При цьому розробники часто стикаються з великим навантаженням через рутинні завдання, такі як створення документації, рефакторинг коду, написання тестів та перевірки коду на відповідності принципам S.O.L.I.D., що може займати значну частину їхнього часу. Згідно з дослідженням [1], рутинні завдання, такі як налагодження коду, можуть займати до 50 % часу розробника, що значно уповільнює процеси розробки.

Постановка проблеми. Наявні інструменти автоматизації, такі як GitHub Copilot, Cody та Tabnine, частково вирішують ці проблеми, однак мають обмеження: вони не забезпечують повноцінної перевірки відповідності принципам S.O.L.I.D., не дозволяють гнучко обирати моделі штучного інтелекту (ШІ) для різних задач і недостатньо інтегровані в середовище розробки для зручного використання результатів. Це призводить до зниження продуктивності розробників і уповільнення випуску продуктів, що підкреслює необхідність розробки інструменту автоматизації, який би інтегрував кілька ШІ-моделей, підтримував принципи якісного програмування та забезпечував зручну взаємодію в середовищі розробки.

Аналіз останніх досліджень і публікацій. Питанням застосування ШІ у розробці ПЗ присвячено значну кількість наукових праць і оглядових досліджень, зокрема [1–7]. Так, у роботі S. Peng та співавторів [1] на основі емпіричних даних з GitHub Copilot досліджено вплив ШІ-асистентів на продуктивність розробників. Автори відзначають зростання ефективності при виконанні рутинних завдань, що підтверджує актуальність використання ШІ-технологій у програмуванні. Огляд екосистеми інструментів для розробників з елементами ШІ наведено в публікації компанії Scott Logic [2], де систематизовано наявні рішення і виявлено ключові напрями розвитку галузі. Подібну тематику розкрито і в огляді N. Upadhyaya [3], в якому описано основні підходи до інтеграції ШІ у життєвий цикл розробки програмного забезпечення. У дослідженні M. Alenezi та M. Akour [4] розглянуто сучасні практики впровадження ШІ в інженерію ПЗ та окреслено перспективи подальших інновацій. Огляд публікацій за результатами досліджень, пов’язаних із використанням ШІ у цій сфері, подано у роботі P. Kokol [5]. Окрему увагу приділено застосуванню ШІ у класичних моделях розробки, що розглядається у статті R.

Sarfraz і T. Riaz [6]. У роботі S. Shankar та S. Chaudhari [7] запропоновано фреймворк автоматизації фаз життєвого циклу ПЗ із використанням технологій ШІ і машинного навчання, що також свідчить про високу зацікавленість наукової спільноти в темі. Очевидно, напрям використання ШІ у розробці ПЗ активно розвивається, і основними цілями досліджень і розробок підвищення продуктивності, автоматизація рутинних процесів, покращення якості програмного коду.

Виділення недосліджених частин загальної проблеми. Новизна розробленого модуля полягає насамперед у гнучкості використання ШІ: на відміну від більшості існуючих рішень, які базуються на одній фіксованій моделі, розроблений Smart AI Code Assistant дозволяє користувачеві самостійно обирати модель для кожного конкретного завдання. Підтримуються GPT, Claude, Gemini, Grok та Deepseek, що дає можливість варіювати вибір залежно від вимог до швидкості, точності або вартості. Це особливо корисно у випадках, коли певна модель гірше справляється з конкретним типом задач – користувач може миттєво перемикнутися на іншу, не змінюючи інтерфейс або середовище розробки.

Ще однією важливою особливістю розробки є підтримка перевірки коду на відповідність принципам S.O.L.I.D. – із поясненням кожного порушення та пропозиціями щодо його усунення. Така функціональність практично не представлена в аналогічних інструментах і значно підвищує якість архітектури коду.

Модуль об’єднує в собі низку ключових функцій: генерування документації, рефакторинг, створення юніт-тестів, пояснення логіки у вигляді коментарів, генерування коду за сигнатурою методу, а також S.O.L.I.D.-аналіз.

Метою дослідження є розробка програмного модуля, який дозволяє, використовуючи популярні ШІ-моделі, автоматизувати ряд рутинних процесів, таких як документування коду, рефакторинг, перевірка на відповідність принципам S.O.L.I.D., генерування модульних тестів та інших функцій для підвищення продуктивності розробників. Для досягнення мети було поставлено такі завдання:

- Провести аналіз існуючих підходів інтелектуальної автоматизації етапів життєвого циклу розробки ПЗ;
- Дослідити можливості ШІ-моделей (ChatGPT, Claude, Gemini, DeepSeek, Grok) у задачах аналізу, оптимізації та генерування програмного коду;
- Розробити методи автоматизованого контролю на відповідність коду принципам S.O.L.I.D.;
- Розробити модуль автоматизованого генерування документації, рефакторингу, модульних тестів, генерування коду за сигнатурою методу, додавання пояснень до коду у вигляді коментарів та перевірки відповідності принципам S.O.L.I.D.;
- На прикладі задачі рефакторингу провести дослідити та оцінити ефективність розробленого інструментарію з погляду прискорення розробки і точності генерування (рівень помилок) для різних моделей.

Виклад основного матеріалу. Для реалізації модуля використано мову програмування JavaScript та платформу Node.js, що дозволило створити легке й масштабоване рішення, інтегроване в популярне середовище розробки.

На етапі проєктування архітектури модуля визначено ключові компоненти: модуль інтеграції з API ШІ-моделей, ядро для обробки запитів користувача (extension.js), компоненти для аналізу коду та генерування результатів, а також інтерфейс для вибору ШІ-моделі. Для інтеграції з ШІ-моделями використано OpenAI API для взаємодії з ChatGPT, а також додано підтримку альтернативних моделей, таких як Claude, Gemini, DeepSeek і Grok, що забезпечило гнучкість у виборі інструменту залежно від задачі. Наприклад, для аналізу S.O.L.I.D. може обиратися Claude через його високу точність, а для швидкого генерування коду – Gemini.

Для аналізу синтаксичної структури коду застосовано бібліотеку tree-sitter, яка забезпечує створення абстрактного синтаксичного дерева (AST) для таких мов програмування, як JavaScript, Java та Python. Це дозволило модулю точно ідентифікувати структуру коду перед обробкою. Бібліотека openai відповідала за взаємодію з API AI-моделей, а tiktoken використовувалася для оптимізації обробки токенів, що зменшило витрати на запити до API. Якщо кількість токенів у коді користувача перевищує максимальний ліміт, який встановлюється в коді програми (наприклад, *const GEMINI_2_5_PRO_LIMIT = 65500;*), то модуль пропонує користувачу виділити меншу частину коду для обробки, запобігаючи обрізанню важливих фрагментів. Конфігураційні параметри, зокрема, API-ключі, зберігалися у файлі .env за допомогою бібліотеки dotenv, що забезпечило безпечне управління чутливими даними, виключаючи їх із системи контролю версій.

Основні функції модуля реалізовано через створення спеціалізованих підмодулів, кожен із яких відповідає конкретному аспекту автоматизації розробки ПЗ.

Генерування документації коду здійснюється шляхом аналізу виділеного фрагмента коду за допомогою ШІ-моделей, які генерують відповідні коментарі для різних мов програмування. Для уникнення небажаних змін у логіці чи структурі коду застосовується безпечний підхід: отримані дані від ШІ очищаються від супутнього форматування, перевіряються на наявність небезпечних функцій (eval, exec тощо) та обробляються через побудову AST. Це дерево дозволяє виділити лише коментарі документації, порівняти їх зі структурою оригінального коду та інтегрувати в потрібні місця без порушення функціональності. Результат відображається у відповідному вікні користувацького інтерфейсу для перегляду й підтвердження.

Для створення документації використано zero-shot промпти, які чітко вказують формат виводу, наприклад, JSDoc, Google-style docstrings, Javadoc, залежно від мови програмування. Промпти містять обмеження на збереження оригінального коду та повернення лише задокументованого результату без додаткового форматування. Нижче подано приклад запиту для задачі документування коду для мов python та java.

```
const documentationPrompts = {
    python: "Generate detailed documentation for the following Python code while keeping the original code unchanged. Use Google-style docstrings and place them directly above the corresponding functions or classes. Each docstring should include a concise description, parameter explanations (Args:), return values (Returns:). Return only the documented code without any additional text or Markdown wrappers.",
    java: "Produce detailed documentation for the following Java code without changing the original code. Add the documentation as Javadoc comments above the relevant classes, methods, or fields. Include descriptions of the class/method purpose, parameters, return types, exceptions, and examples. Return only the documented code without any additional text or Markdown wrappers.",
    default: "Generate detailed documentation for the following code without modifying the original code. Add the documentation as comments above the relevant sections, including a description of its purpose, inputs, outputs, and a simple usage example. Return only the documented code without any additional text or Markdown wrappers."
};
```

Рефакторинг коду. Процес ініціюється через команду, яка визначає контекст коду та відправляє його до ШІ для аналізу і покращень, таких як розбиття на логічні блоки чи використання потрібних конструкцій і шаблонів. При цьому система автоматично обирає відповідну ШІ-модель залежно від обсягу коду. Користувач може затвердити зміни, скасувати їх або запросити альтернативний варіант, що додає гнучкості щодо різних потреб розробки.

Промпти для рефакторингу орієнтовані на покращення читабельності, продуктивності та відповідності стандартам, наприклад, ES6+ для JavaScript, PEP 8 для Python. Використовується підхід zero-shot із обмеженнями на збереження функціональності та структури. Убезпечення від появи небажаних функцій, наприклад, eval чи exec, враховується при формуванні звернення до ШІ-рушія.

Генерування тестів та їх інтеграція у проєкт передбачає створення юніт-тестів на основі аналізу коду за допомогою ШІ, з урахуванням типових і граничних випадків. Після генерування тестів користувачу пропонується обрати місце їх збереження, включаючи можливість створення нового файлу чи використання існуючого. Важливою особливістю є автоматичне злиття з уже наявними тестами: система виконує аналіз поточного вмісту тестового файлу, парсить код за допомогою бібліотеки *tree-sitter*, виявляє наявні тестові функції або методи, порівнюючи їхні назви чи описи зі згенерованими, і додає лише ті нові тести, які ще не присутні у файлі. Таким чином, уникається дублювання, а нові тести інтегруються у логічно відповідні місця, з урахуванням структури коду та синтаксису конкретної мови програмування. Це дозволяє розширювати покриття тестів без конфліктів і додаткових зусиль.

Промпти для створення юніт-тестів (Jest/Mocha для JavaScript, unittest/pytest для Python, JUnit для Java) включають вимоги до покриття основної функціональності, граничних випадків і обробки помилок.

Аналіз коду на відповідність принципам S.O.L.I.D. забезпечує оцінку коду за ключовими принципами об’єктно-орієнтованого дизайну, виявляючи потенційні порушення. Процес включає автоматичну перевірку при зміні чи відкритті файлу з урахуванням оптимального навантаження, а також ручне ініціювання детального аналізу. Результати відображаються в редакторі з підкресленням проблемних ділянок і короткими рекомендаціями, доступними при наведенні курсору. Користувач може запросити покрокові інструкції для виправлення або отримати автоматично запропоновані рішення, які інтегруються як швидкі дії, що значно полегшує покращення якості коду.

Промпти реалізують трирівневий підхід: аналіз із виводом порушень у JSON-форматі, рекомендації щодо їх усунення та автоматичне генерування відрефакторeного коду з обмеженнями на збереження залежностей і фокусом на цільовому класі.

Реалізація вибору ШІ-моделі здійснена через інтерфейс у Visual Studio Code, де користувач обирає модель: ChatGPT, Claude, Gemini, DeepSeek або Grok у веб-перегляді, налаштовуючи її для кожної задачі.

Особливу увагу приділено оптимізації продуктивності та стабільності. Проведено ітерації тестування для виправлення помилок, пов’язаних із великими файлами чи некоректними відповідями ШІ. У результаті створено інструмент, який дозволяє автоматизувати рутинні завдання, підвищити продуктивність і якість коду.

Налаштування та запуск модуля. Для початку роботи з розширенням його необхідно встановити в середовищі Visual Studio Code, яке доступне як через редактор, так і через Visual Studio Marketplace. Установка виконується шляхом відкриття вкладки Extensions (Ctrl+Shift+X), введення назви «Smart AI Code Assistant» у пошуку, вибору розширення від автора AI-Assistant (рис. 1), та натискання кнопки Install.

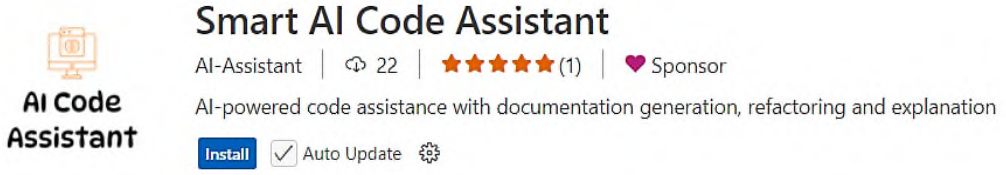

Рис. 1. Встановлення розширення у Visual Studio Code

Після завершення процесу розширення активується автоматично. Встановлення модуля можна також здійснити через Marketplace за посиланням: https://marketplace.visualstudio.com/items?itemName=AI-Code-Assistant.smart-ai-code-assistant. Після успішного завершення розширення з'явиться в списку встановлених. Налаштування параметрів доступне через меню Settings (Ctrl+,). Основні опції:

- aiCodeAssistant.defaultModel – вибір ШІ-моделі за замовчуванням (наприклад, grok-2-latest);
- aiCodeAssistant.enableAutoSolidAnalysis – увімкнення/вимкнення автоматичного аналізу S.O.L.I.D. (за замовчуванням вимкнено);
- aiCodeAssistant.debugMode – активація режиму налагодження (за замовчуванням вимкнено).

Для коректної роботи потрібна версія Visual Studio Code 1.97.0 або новіша. Залежності, такі як tree-sitter, openai та anthropic-ai/sdk, встановлюються автоматично, забезпечуючи повну функціональність. Детальний опис можливостей, команд та налаштувань доступний у вкладці Details у VS Code або Marketplace, що дозволяє ознайомитися з інструментом до встановлення.

Приклади використання. Генерування документації продемонстровано на прикладі Python-функції calculate_order_total без початкової документації. Користувач виділяє код і активує відповідну функцію (рис. 2).

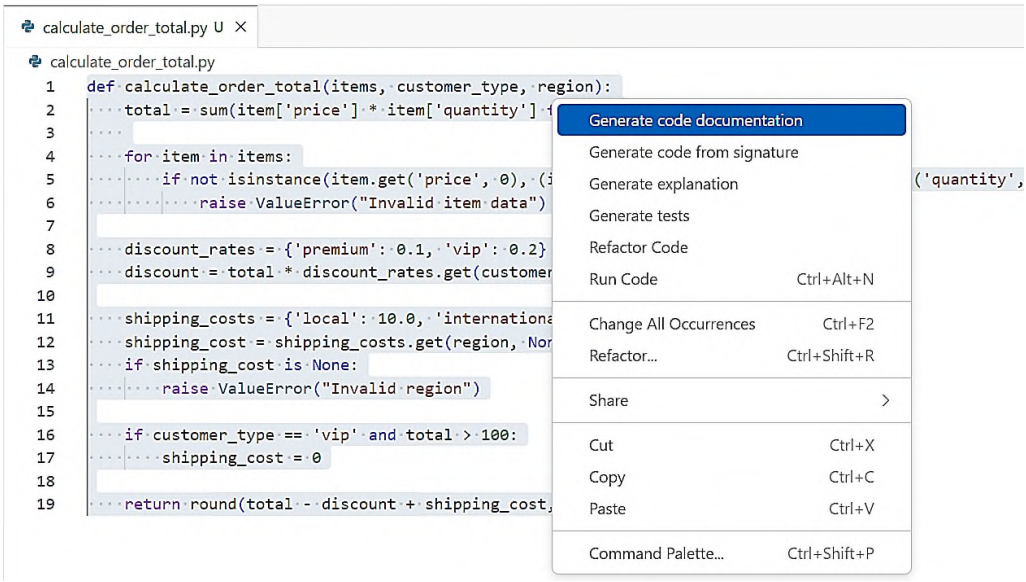

Рис. 2. Виділення коду та вибір функції генерування документації

Результат відображається у вебперегляді: зліва – оригінальний код, справа – код із доданою docstring-документацією (рис. 3).

Після затвердження оновлений код автоматично інтегрується у файл, гарантуючи незмінність коду користувача, завдяки розумній вставці документації, описаній вище. У підсумку отримано docstring-коментарі, що описують методи, їхні параметри та повернені значення, полегшуючи розуміння логіки для інших розробників.

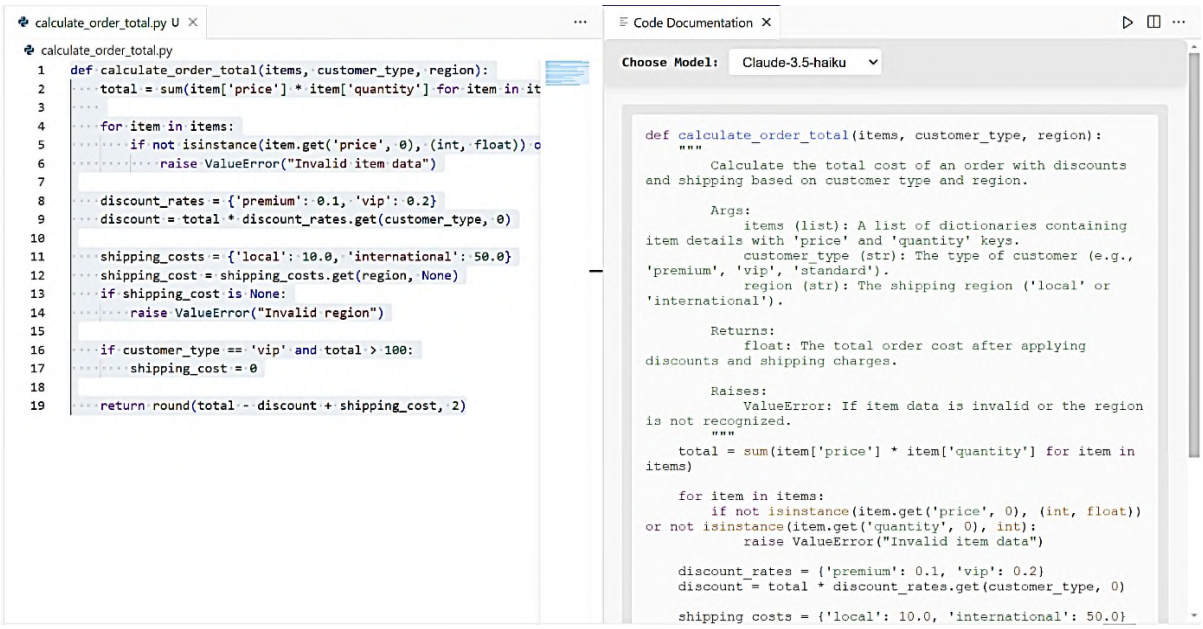

Рис. 3. Результат генерування документації

Автоматизацію рефакторингу продемонстровано на прикладі JavaScript-коду з проблемами читабельності, де функція в одному блоці обробляє список користувачів, фільтрацію, ролі, статус підписки та логування, ускладнюючи розуміння та модифікацію. Користувач виділяє код і активує функцію рефакторингу (рис. 4).

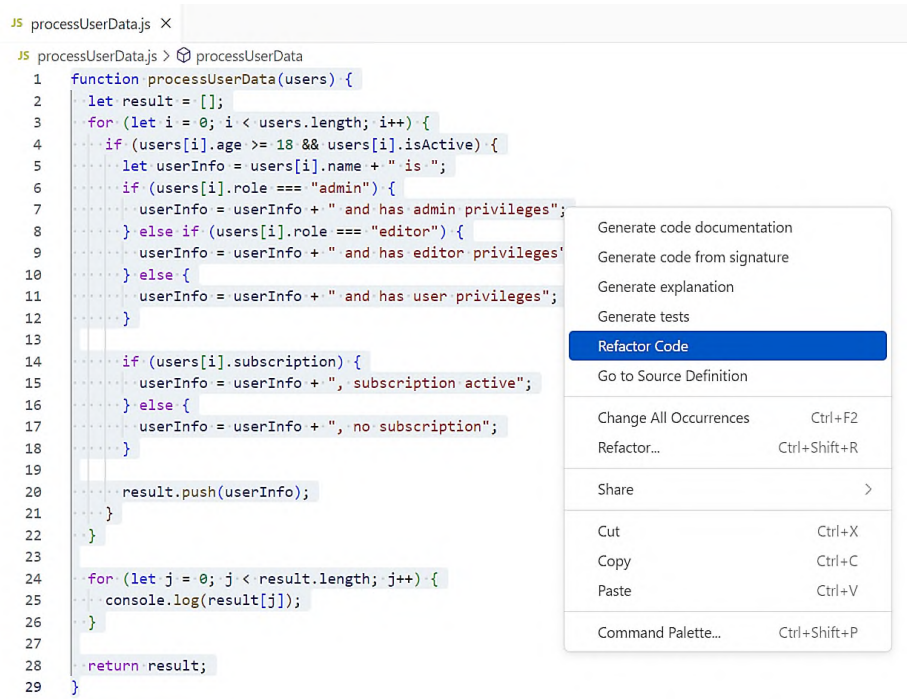

Рис. 4. Виділення коду та вибір функції рефакторингу

У вебперегляді результат показує: зліва – оригінальний код, справа – відрефакторена версія з покращеною структурою, використанням методів filter, map, forEach і логічним поділом (рис. 5). Після затвердження змін оновлений код замінює початковий, стаючи більш модульним, читабельним і стійким до помилок.

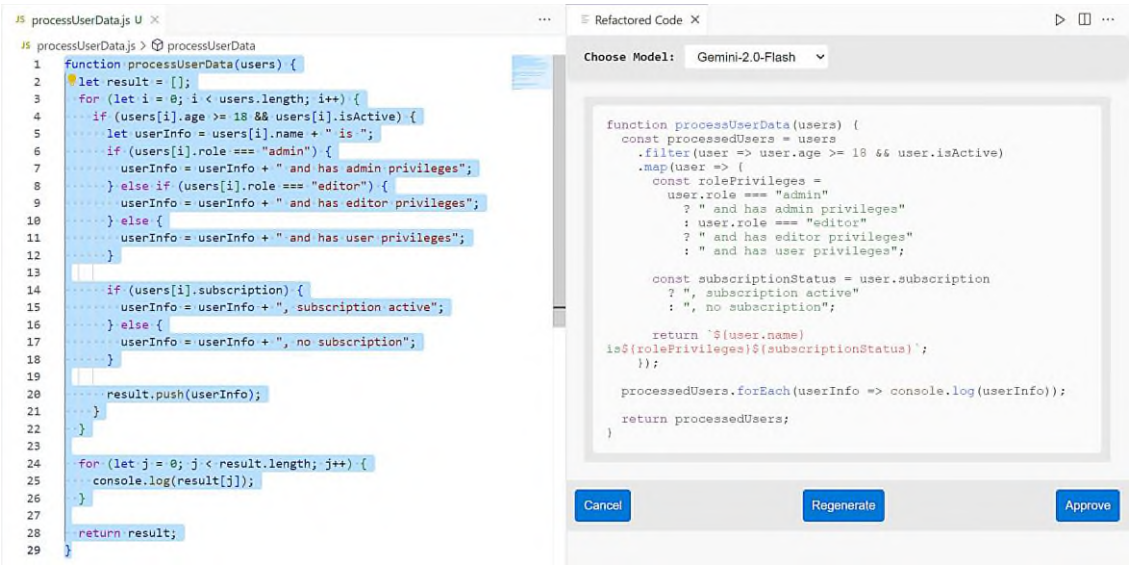

Рис. 5. Відрефакторений код, згенерований розширенням

Функціонал перевірки коду на відповідність принципам S.O.L.I.D. продемонстровано на прикладі Java-коду, що порушує принцип єдиної відповідальності (SRP). При відкритті файлу автоматично запускається аналіз, про що сповіщає повідомлення «S.O.L.I.D.-аналіз активовано» у нижній панелі редактора (рис. 6).

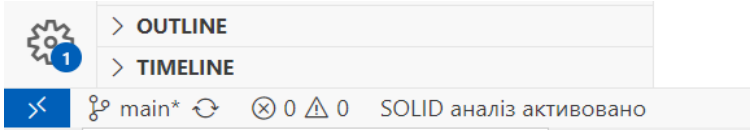

Рисунок 6 – Повідомлення про активацію S.O.L.I.D.*-аналізу*

Після завершення аналізу розширення виділяє назву класу з виявленим порушенням. При наведенні курсора з'являється спливаюче повідомлення з описом проблеми та короткою рекомендацією щодо її виправлення (рис. 7).

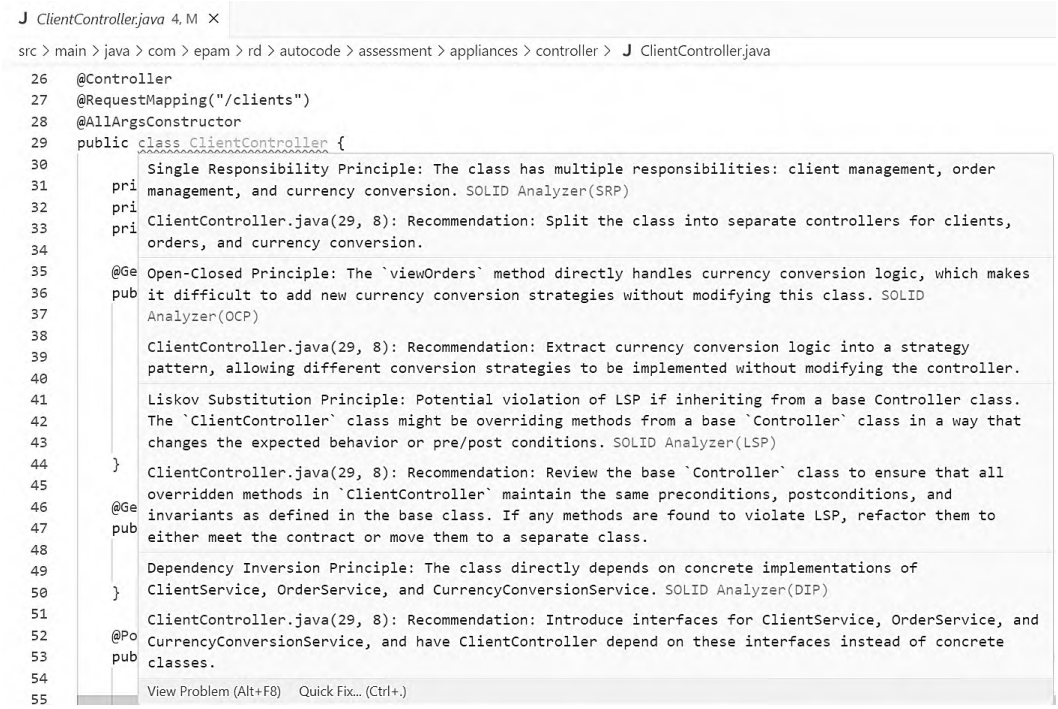

Рис. 7. Підкреслення класу з порушенням та рекомендації при наведенні

Користувач може обрати опцію Quick Fix, що відкриває панель із командами: отримати детальні рекомендації або згенерувати виправлений код для відповідності принципам (рис. 8).

Рис. 8. Панель із командами після вибору Quick Fix

Вибір опції детальних рекомендацій відкриває веб-перегляд із покроковою інструкцією, як усунути порушення для відповідності S.O.L.I.D. (рис. 9). Якщо ж обрати команду генерування виправленого коду, розширення представить оновлену версію, де порушення усунуто шляхом розподілу логіки між окремими класами відповідно до SRP.

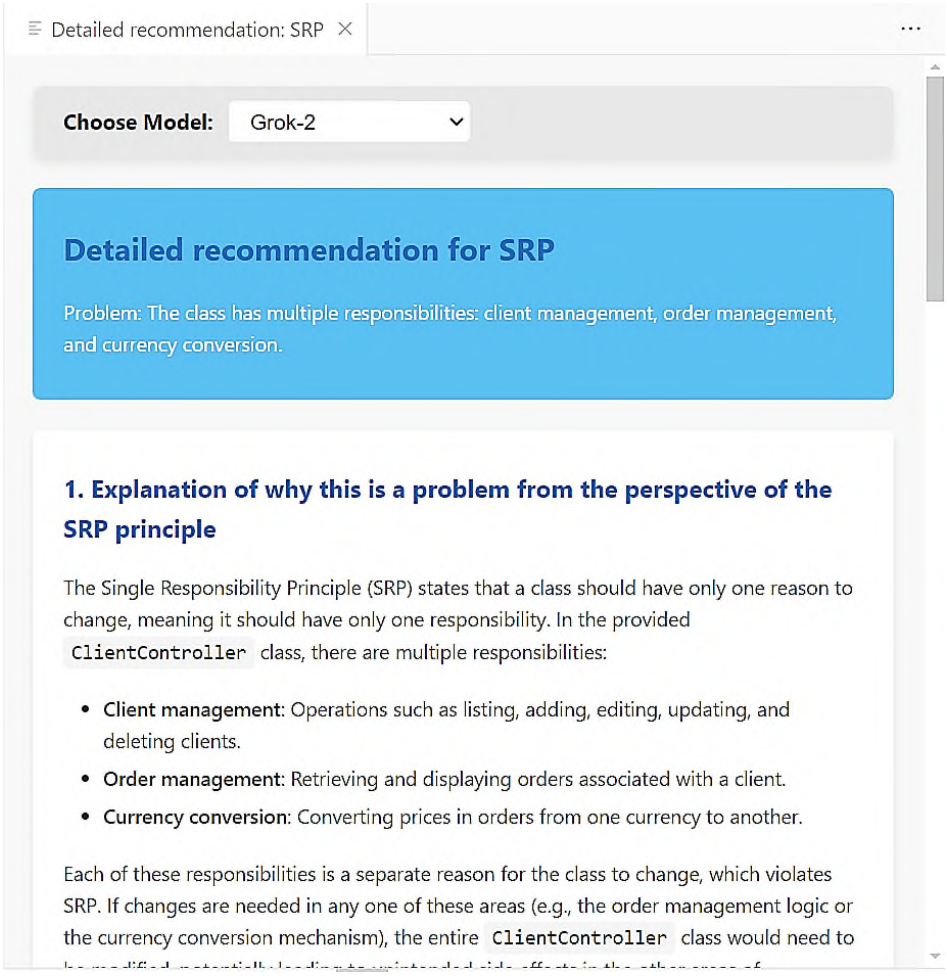

Рис. 9. Покрокові рекомендації щодо покращення коду

Дослідження ефективності ШІ-моделей у задачі автоматизації рефакторингу коду. Для оцінки ефективності було проведено дослідження на основі аналізу якості результатів на модульних тестах. Методика полягала у створенні тестів для вхідного коду, що підлягає рефакторингу, з подальшою перевіркою їх проходження після рефакторингу кожною з моделей. Такий підхід дозволив оцінити рівень виникнення помилок та час, необхідний моделі для виконання завдання. Для досягнення статистично достовірних результатів експеримент проводився на 100 спробах для кожної з моделей.

Прикладом для демонстрації рефакторингу обрано клас TaskManager, реалізований мовою JavaScript, який відповідає за управління списком задач. Клас має низку типових проблем, що ускладнюють підтримку: неінформативні скорочення змінних (напр. dl, cr, n, d), відсутність розподілу відповідальностей через централізовану архітектуру, неефективні ручні цикли замість методів filter чи find, а також дублювання логіки у фільтрації та пошуку. Ці недоліки роблять TaskManager показовим прикладом для демонстрації покращень через рефакторинг.

Експериментальна установка працює так: оригінальний код із файлу *TaskManager.js*, який містить типові проблеми, передається кожній ШІ-моделі з чітким запитом на рефакторинг. У запиті зазначається заборона змінювати назви методів, аби зберегти сумісність із наявними тестами. Досліджувалися такі моделі: gpt-4-turbo, gpt-3.5-turbo, gpt-4o, grok-3-latest, grok-2-latest, deepseek-chat, claude-3-7-sonnet-20250219, claude-3-5-haiku-20241022, gemini-2.5-pro-exp-03-25, gemini-1.5-pro-latest і gemini-2.0-flash. Відрефакторений код зберігається у файл *TaskManager.js*, після чого автоматично запускаються тести. Фіксуються кількість успішних і провалених тестів, а також час відповіді моделі. Результати 100 запусків для кожної моделі наведено на рис. 10.

Model	Success Rate	Avg Response Time	Passed	Failed	Total
deepseek-chat	100.00%	33.14s	3200	0	3200
claude-3-7-sonnet-20250219	100.00%	10.81s	3200	0	3200
grok-3-latest	99.93%	16.55s	2974	2	2976
claude-3-5-haiku-20241022	99.81%	14.26s	3194	6	3200
gpt-4o	99.66%	7.67s	3189	11	3200
gpt-3.5-turbo	99.16%	7.64s	3173	27	3200
grok-2-latest	99.13%	8.48s	3172	28	3200
gemini-2.0-flash	97.75%	4.24s	3128	72	3200
gemini-1.5-pro-latest	94.87%	9.75s	1184	64	1248
gpt-4-turbo	71.53%	21.44s	2289	911	3200
gemini-2.5-pro-exp-03-25	69.27%	30.33s	133	59	192

Рис. 10. Результати вимірювання ефективності ШІ-моделей у задачі автоматизації рефакторингу коду

Як показано на рис. 10-11, найкращі результати продемонстрували моделі deepseek-chat та claude-3-7-sonnet-20250219, які досягли 100 % проходження тестів без помилок. Це демонструє їх здатність рефакторити код без втрати функціональності краще, ніж інші моделі. Водночас claude-3-7-sonnet перевершує deepseek-chat за швидкістю: 10.81 секунди проти 33,14, що робить її більш зручною для інтерактивних сценаріїв.

Високу стабільність продемонстрували grok-3-latest (99,93%), claude-3-5-haiku (99,81 %) та gpt-4o (99.66%) із хорошим балансом між точністю та швидкістю (7,67–19,88 с). gpt-3.5-turbo (99.16%) і grok-2-latest (99,13 %) також показали високі результати при ще швидшій роботі (7,64–8,48 с). Як видно з рис. 11, gemini-2.0-flash (97,75 %, 4,24 с) вирізняється швидкістю, але меншою стабільністю. Найгірші результати мали gpt-4-turbo (71,53 %) і gemini-2.5-pro-exp (69,27 %), остання виконала лише 192 з 3200 тестів через проблеми з компіляцією згенерованого коду в інших 3008 випадках. Це свідчить про недостатню стабільність дослідженої версії моделі у забезпеченні синтаксичної коректності для даної задачі. Gemini-1.5-pro показала середню якість (94,87 %) і мала кількість запусків, нижчу відносно інших (1248) з аналогічної причини.

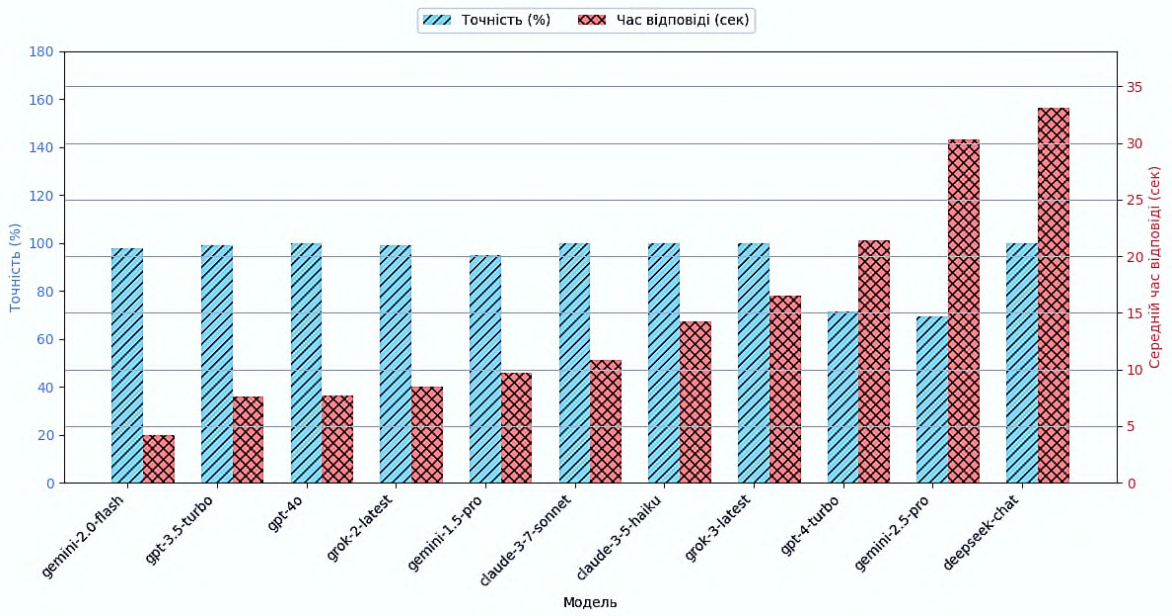

Рис. 11. Результати експерименту, відсортовані за середнім часом відповіді ШІ-моделей

Для інтегральної оцінки ШІ-моделей було введено метрику Score, яка враховує як точність рефакторингу (successRate), так і середній час відповіді (avgTime). Формула:

$$Score = \frac{k_1 \times successRate}{100} + \frac{k_2}{avgTime},$$

де *successRate* – це відсоток успішного проходження тестів;

avgTime – середній час відповіді моделі в секундах;

k_1, k_2 – коефіцієнти, що визначають вагу точності та швидкодії, у нашому випадку обрано 70 і 30 відповідно.

Відповідно до отриманих результатів, найвищий Score отримала gemini-2.0-flash (75,51 бала) завдяки відносно швидкій роботі (4,24 с) при високій точності (97,75 %) (рис. 12). Високу ефективність також продемонстрували gpt-4o (73,67) та gpt-3.5-turbo (73,34), обидві поєднують високу стабільність зі швидкою відповіддю. Якщо ж головним критерієм є саме якість рефакторингу, найкращий вибір – claude-3-7-sonnet або deepseek-chat, які показали 100% успішності, хоч і працюють повільніше.

Model	Success Rate	Avg Response Time	Score
gemini-2.0-flash	97.75%	4.24s	75.51
gpt-4o	99.66%	7.67s	73.67
gpt-3.5-turbo	99.16%	7.64s	73.34
grok-2-latest	99.13%	8.48s	72.93
claude-3-7-sonnet-20250219	100.00%	10.81s	72.77
claude-3-5-haiku-20241022	99.81%	14.26s	71.97
grok-3-latest	99.93%	16.55s	71.77
deepseek-chat	100.00%	33.14s	70.91
gemini-1.5-pro-latest	94.87%	9.75s	69.49
gpt-4-turbo	71.53%	21.44s	51.47
gemini-2.5-pro-exp-03-25	69.27%	30.33s	49.48

Рис. 12. Порівняльна інтегральна оцінка ефективності ШІ-моделей за результатами експерименту

Результати випробувань (рис. 10-12) демонструють різну ефективність ШІ-моделей у задачі рефакторингу коду, якщо оцінювати точність і швидкість обробки. Для більш повної оцінки розглянутих моделей [8-12] варто відзначити додаткові характеристики, зокрема, розмір контекстного вікна, максимальну кількість вхідних і вихідних токенів, вартість використання API (табл. 1), що може вплинути на вибір моделей розробниками з урахуванням їхніх потреб та можливостей.

Можемо побачити, що моделі Gemini 2.5 Pro та Gemini 2.0 Flash вирізняються найбільшим контекстним вікном – 1M токенів, що робить їх оптимальними для обробки великих обсягів коду чи документації. Claude 3.7 Sonnet і Claude 3.5 Haiku (200K) та GPT-4 Turbo/GPT-4o (128K) також підтримують значні обсяги контексту, але поступаються за вихідними токенами моделям Grok 3, Grok 2 та Gemini 2.5 Pro. З економічного погляду, Gemini 2.0 Flash, який коштує 0,10-0,40 $ за 1M токенів та DeepSeek R1, який коштує 0,27-1,10 $, є найвигіднішими, тоді як GPT-4 Turbo має найвищу вартість: $10.00-$30.00.

Враховуючи якісні та вартісні показники, можна відзначити модель gemini-2.0-flash як один із можливих оптимальних виборів.

Таблиця 1 – Характеристики та вартість використання AI-моделей за API

Модель	**Контекстне вікно**	**Вхідні токени (макс.)**	**Вихідні токени (макс.)**	**Ціна вхід ($/1M токенів),** $	**Ціна вихід ($/1M токенів),** $
GPT-4 Turbo	128K	128 000	4 096	10.00	30.00
GPT-4o	128K	128 000	16 384	2.5	10
Grok 3	131K	65 536	65 536	3.00	15.00
Grok 2	131K	65 536	65 536	2.00	10.00
DeepSeek R1	64K	56 000	8 192	0.27	1.10
Claude 3.7 Sonnet	200K	200 000	8 192	3.00	15.00
Claude 3.5 Haiku	200K	200 000	8 192	0.80	4.00
Gemini 2.5 Pro	1M	1 048 576	65 536	1.25	10.00
Gemini 2.0 Flash	1M	1 048 576	8 192	0.10	0.40

Висновки. Розроблено новий модуль розширення Smart AI Code Assistant для Visual Studio Code, що забезпечує часткову автоматизацію процесів розробки ПЗ. Він реалізує функції автоматичного генерування документації до коду, його рефакторингу, створення модульних тестів, генерування коду за сигнатурою методу, пояснення логіки у вигляді коментарів, а також перевірки відповідності коду принципам S.O.L.I.D. з можливістю автоматичного виправлення. Особливістю і новизною є можливість обирати модель ШІ в процесі розробки. Використання ШІ-моделей, таких як ChatGPT, Grok, Claude, DeepSeek і Gemini, у поєднанні з модульною архітектурою забезпечило гнучкість і масштабованість рішення. Тестування на реальних прикладах, таких як функція calculate_order_total для документування чи JavaScript-код для рефакторингу, підтвердило ефективність розширення щодо підвищення продуктивності розробників і покращенні якості коду. Автоматичне злиття тестів і безпечна вставка документації через аналіз структури коду демонструють надійність підходу. Розроблене розширення готове до практичного використання.

До **перспективних напрямів подальшої розробки** можна віднести додавання підтримки нових мов програмування, таких як C#, C++, Go і TypeScript, інтеграція з платформами CI/CD, зокрема Jenkins, GitHub Actions або GitLab CI, для автоматизації перевірки S.O.L.I.D., генерування тестів і документації під час комітів чи pull-request-ів, додавання автоматичного code review в рамках pull-request-ів. Передбачається інтеграція з інструментами аналізу якості коду, такими як SonarQube, ESLint і Checkstyle, для комбінування ШІ-аналізу з традиційними статичними методами.

Заява про використання генеративного ШІ та технологій на основі ШІ в процесі написання тексту статті.

Під час написання цього матеріалу автори не використовували інструменти генеративного штучного інтелекту.

Список використаних джерел

1. Peng S., Kalliamvakou E., Cihon P., Demirer M. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot, *arXiv:2302.06590*, 13 лют. 2023.

2. Making sense of the AI developer tools ecosystem. (n.d.). *Scott Logic*. https://blog.scottlogic.com/2025/04/01/making-sense-of-the-ai-developer-tools-ecosystem.html.

3. Upadhyaya, N. (n.d.). *The role of artificial intelligence in software development: A literature review*. https://www.researchgate.net/profile/Nitesh-Upadhyaya-2/publication/385781688_The_Role_of_Artificial_Intelligence_in_Software_Development_A_Literature_Review/links/67350a6869c07a4114468292/The-Role-of-Artificial-Intelligence-in-Software-Development-A-Literature-Review.pdf.

4. Alenezi, M., & Akour, M. (2025). AI-Driven innovations in software engineering: A review of current practices and future directions. *Applied Sciences*, *15*(3), 1344. https://doi.org/10.3390/app15031344.

5. Kokol, P. (2024). The use of AI in software engineering: A synthetic knowledge synthesis of the recent research literature. *Information*, *15*(6), 354. https://doi.org/10.3390/info15060354.

6. Sarfraz, Rawish & Riaz, Tehreem. (2024). Applications of AI in Classical Software Engineering. https://link.springer.com/article/10.1186/s42467-020-00005-4.

7. Shankar, S. P., & Chaudhari, S. S. (2023). Framework for the automation of SDLC phases using artificial intelligence and machine learning techniques. *International Journal on Recent and Innovation Trends in Computing and Communication*, *11*(6s), 379–390. https://doi.org/10.17762/ijritcc.v11i6s.6944.

8. OpenAI. (n.d.). *OpenAI API models*. OpenAI – official website. https://platform.openai.com/docs/models.

9. Models and Pricing. xAI Documentation – official website. https://docs.x.ai/docs/models?cluster=us-east-1.

10. Models & Pricing. DeepSeek API Docs – official website. https://api-docs.deepseek.com/quick_start/pricing/.

11. Models & pricing. Anthropic – official website. https://docs.anthropic.com/en/docs/about-claude/models/all-models.

12. Gemini Developer API Pricing. Google AI for Developers – official website. https://ai.google.dev/gemini-api/docs/pricing.

References

1. Peng S., Kalliamvakou E., Cihon P., Demirer M. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot, *arXiv:2302.06590*, 13 лют. 2023.

2. Making sense of the AI developer tools ecosystem. (n.d.). *Scott Logic*. https://blog.scottlogic.com/2025/04/01/making-sense-of-the-ai-developer-tools-ecosystem.html.

3. Upadhyaya, N. (n.d.). *The role of artificial intelligence in software development: A literature review*. https://www.researchgate.net/profile/Nitesh-Upadhyaya-2/publication/385781688_The_Role_of_Artificial_Intelligence_in_Software_Development_A_Literature_Review/links/67350a6869c07a4114468292/The-Role-of-Artificial-Intelligence-in-Software-Development-A-Literature-Review.pdf.

4. Alenezi, M., & Akour, M. (2025). AI-Driven innovations in software engineering: A review of current practices and future directions. *Applied Sciences*, *15*(3), 1344. https://doi.org/10.3390/app15031344.

5. Kokol, P. (2024). The use of AI in software engineering: A synthetic knowledge synthesis of the recent research literature. *Information*, *15*(6), 354. https://doi.org/10.3390/info15060354.

6. Sarfraz, Rawish & Riaz, Tehreem. (2024). Applications of AI in Classical Software Engineering. https://link.springer.com/article/10.1186/s42467-020-00005-4.

7. Shankar, S. P., & Chaudhari, S. S. (2023). Framework for the automation of SDLC phases using artificial intelligence and machine learning techniques. *International Journal on Recent and Innovation Trends in Computing and Communication*, *11*(6s), 379–390. https://doi.org/10.17762/ijritcc.v11i6s.6944.

8. OpenAI. (n.d.). *OpenAI API models*. OpenAI – official website. https://platform.openai.com/docs/models.

9. Models and Pricing. xAI Documentation – official website. https://docs.x.ai/docs/models?cluster=us-east-1.

10. Models & Pricing. DeepSeek API Docs – official website. https://api-docs.deepseek.com/quick_start/pricing/.

11. Models & pricing. Anthropic – official website. https://docs.anthropic.com/en/docs/about-claude/models/all-models.

12. Gemini Developer API Pricing. Google AI for Developers – official website. https://ai.google.dev/gemini-api/docs/pricing.

Отримано 19.06.2025

UDC 004.41

Oleksii Kondus[1], Oleksii Tkachenko[2]

[1]master
Taras Shevchenko National University of Kyiv (Kyiv, Ukraine)
E-mail: oleksii.kondus@knu.ua. **ORCID:** https://orcid.org/0009-0003-2514-6196. **ResearcherID:** NLN-6147-2025

[2] PhD in Computer Sciences, Associate Professor, Department of Theory and Technology of Programming
Taras Shevchenko National University of Kyiv (Kyiv, Ukraine)
E-mail: otkachenko@knu.ua. **ORCID:** https://orcid.org/0000-0002-9514-516X. **ResearcherID:** U-3613-2017

INTELLECTUALIZED SUPPORT MODULE OF THE SOFTWARE DEVELOPMENT

The fast-paced information technology market demands rapid and high-quality software development to stay competitive. However, routine tasks such as code documentation, refactoring, test creation, and ensuring S.O.L.I.D. principle compliance consume a significant amount of developer time, with studies showing that up to 50% of effort is spent on such activities. Existing tools, such as GitHub Copilot and Tabnine, offer partial automation but lack comprehensive S.O.L.I.D. analysis, flexible AI model selection, and seamless integration within environments like Visual Studio Code. This highlights the need for a robust solution to streamline workflows, aligning with the growing use of AI assistants to boost coding efficiency and quality.

This study tackles these challenges by introducing the Smart AI Code Assistant, a VS Code extension that automates routine tasks using AI models such as GPT, Claude, Gemini, Grok, and DeepSeek. The research aims to enhance developer productivity through automated documentation, refactoring, unit test generation, and S.O.L.I.D. compliance checks within a unified interface. Unlike other tools, the module allows task-specific AI model selection for optimal speed, accuracy, and cost, and provides detailed S.O.L.I.D. analysis with actionable feedback, improving code architecture.

The methodology involved analyzing automation trends, evaluating AI model capabilities, and developing S.O.L.I.D. verification methods. Built with JavaScript and Node.js, the module uses Tree-sitter for code analysis and supports languages like JavaScript, Java, and Python. Key features include safe documentation generation, modular refactoring, test integration, and S.O.L.I.D. violation reports with fixes. Experiments tested refactoring performance on a flawed JavaScript TaskManager class across 100 trials, assessing test pass rates and response times.

The results of the experiments demonstrated varying effectiveness of AI models. Claude-3-7-sonnet and deepseek-chat achieved 100% test pass rates, with Claude faster (10.81 vs. 33.14 seconds). Gemini-2.0-flash balanced speed (4.24 seconds) and accuracy (97.75%), offering cost-effectiveness. The module's cohesive VS Code integration reduces manual effort and enhances code quality. It is a practical tool with potential for expansion to languages like C# and Go, and CI/CD integration. The Smart AI Code Assistant advances software development by addressing existing tool limitations, enabling faster, higher-quality outputs.

Keywords: *automation of software development; artificial intelligence; S.O.L.I.D.; refactoring; code documentation; AI models; Visual Studio Code.*

Fig.: 12. Table: 1. References: 12.

Кондус О. С., Ткаченко О. М. Модуль інтелектуалізованої підтримки розробки програмного забезпечення. *Технічні науки та технології.* 2025. № 2(40). С. 312-324. DOI: https://doi.org/10.25140/2411-5363-2025-2(40)-312-324.